Hakin9

PRACTICAL PROTECTION

IT SECURITY MAGAZINE

Vol.9 No.02
Issue 02/2014(71) ISSN: 1733-7186

REVERSE ENGINEERING

PLAYING WITH THE PORTS REDIRECTIONS

REVERSING CRACKIN

MALWARE AN

WRITE YOUR OWN DEBUGGER

S LOGIC

TEASER

ISSE 2014

14th & 15th October 2014
MCE, Brussels, Belgium
www.isse.eu.com

Securing Assets Across Europe

Europe's leading independent, interdisciplinary security conference and exhibition

Over the past decade, Information Security Solutions Europe (ISSE) has built an unrivalled reputation for its world-class, interdisciplinary approach and independent perspective on the e-security market.

This year, ISSE will take place on 14th & 15th October in Brussels. Regularly attracting over 300 professionals including government, commercial end-users and industry experts who will come together for a unique all-encompassing opportunity to learn, share and discuss the latest developments in e-security and identity management.

Programme Topic Areas

- **Trust Services, eID and Cloud Security**
  European trust services and eldentity regulation, governance rules, standardization, interoperability of services and applications, architectures in the cloud, governance, risks, migration issues
- **BYOD and Mobile Security**
  Processes and technologies for managing BYOD programs, smartphone/tablet security, mobile malware, application threats
- **Cybersecurity, Cybercrime, Critical Infrastructures**
  Attacks & countermeasures against industrial Infrastructures; CERT/CSIRT – European & global developments, resilience of networks & services, surveillance techniques & analytics
- **Security Management, CISO Inside**
  CISOs featuring the latest trends and issues in information security, risk mitigation, compliance & governance; policy, planning and emerging areas of enterprise security architecture
- **Privacy, Data Protection, Human Factors**
  Issues in big data & cloud, privacy enhancing technologies, insider threats, social networking/engineering and security awareness programs
- **Regulation & Policies**
  Governmental cybersecurity strategies, authentication, authorization & accounting, governance, risk & compliance

In partnership with eema www.eema.org TeleTrusT Pioneers in IT security.

For more information visit www.isse.eu.com

@ISSEConference

Reverse Engineering

Table of Contents

standard until evolving into TLS. TLS/SSL uses the public-and-private key encryption system from RSA, which also includes the use of a digital certificate. TLS/SSL is an integral part of most Web browsers (the client side) and Web servers.

Dear Hakin9 Readers!

We give you the new Hakin9 issue about Reverse Engineering. Less theory, more practice!

What will you find here?

First of all you will find out WHAT is Reverse Engineering.

You will also learn about: How does software break? Why hackers and crackers easily break into your secure systems? Why web applications got hacked every second day?

Or maybe you want know how to detect systems that are vulnerable to the OpenSSLHeartbleed vulnerability and learn how to exploit them using Metasploit on Kali Linux?

And if you want to write your own debugger then the article by Amr Thabet ‘Write your own debugger’ is definitely for you!

You will learn what is required in order to start thinking as a cracker.

And MORE in the latest issue of Hakin9 Magazine!

Wish you a good reading!

Hakin9 Magazine Team

Editor in Chief: Ewa Dudzic
ewa.dudzic.@hakin9.org

Managing Editor: Patrycja Wąsik
patrycja.wasik@hakin9.org

Executive Editor: Karolina Chrostna-Bejnarowicz
karolina.chrostna-bejnarowicz@hakin9.org

Advisory Board Member: Luca Ferrari

Editorial Advisory Board: Jeff Smith, Craig Thornton, Elia Pinto

Betatesters & Proofreaders: David, Kosorok, Tom Updegrove, Craig Thornton, Jackson Bennett, Elia Pinto, Jeff Smith, David von Vistauxx, Gilles Lami, Julian Esteves, K. S. Abhiraj, Arnoud Tijssen, Shahid Hussain Rathore, Kishore Rathav

Special Thanks to the Beta testers and Proofreaders who helped us with this issue. Without their assistance there would not be a Hakin9 Magazine.

Publisher: Paweł Marciniak

CEO: Ewa Dudzic
ewa.dudzic.@hakin9.org

Art. Director: Ireneusz Pogroszewski
ireneusz.pogroszewski@hakin9.org
DTP: Ireneusz Pogroszewski

Publisher: Hakin9 Media sp. z o.o. SK
02-676 Warszawa, ul. Postępu 17D
NIP 95123253396
www.hakin9.org/en

What is Reverse Engineering

by Aman Singh

Reverse engineering as this article will discuss it is simply the act of figuring out what software that you have no source code for does in a particular feature or function to the degree that you can either modify this code, or reproduce it in another independent work.

In the general sense, ground-up reverse engineering is very hard, and requires several engineers and a good deal of support software just to capture the all of the ideas in a system. However, we'll find that by using tools available to us, and keeping a good notebook of what's going on, we should be able to extract the information we need to do what matters: make modifications and hacks to get software that we do not have source code for to do things that it was not originally intended to do.

Why Reverse Engineer?

Answer: Because you can.

It comes down to an issue of power and control. Every computer enthusiast (and essentially any enthusiast in general) is a control-freak. We love the details. We love being able to figure things out. We love to be able to wrap our heads around a system and be able to predict its every move, and more, be able to direct its every move. And if you have source code to the software, this is all fine and good. But unfortunately, this is not always the case.

Furthermore, software that you do not have source code to is usually the most interesting kind of software. Sometimes you may be curious as to how a particular security feature works, or if the copy protection is really "unbreakable", and sometimes you just want to know how a particular feature is implemented.

It makes you a better programmer

This article will teach you a large amount about how your computer works on a low level, and the better an understanding you have of that, the more efficient programs you can write in general.

To Learn Assembly Language

If you don't know assembly language, at the end of this article you will literally know it inside-out. While most first courses and articles on assembly language teach you how to use it as a programming language, you will get to see how to use C as an assembly language generation tool, and how to look at and think about assembly as a C program. This puts you at a tremendous advantage over your peers not only in terms of programming ability, but also in terms of your ability to figure out how the black box works. In short, learning this way will naturally make you a better reverse engineer. Plus, you will have the fine distinction of being able to answer the question "Who taught you assembly language?" with "Why, my C compiler, of course!"

Intro

Compilation in general is split into roughly 5 stages: Preprocessing, Parsing, Translation, Assembling, and Linking.

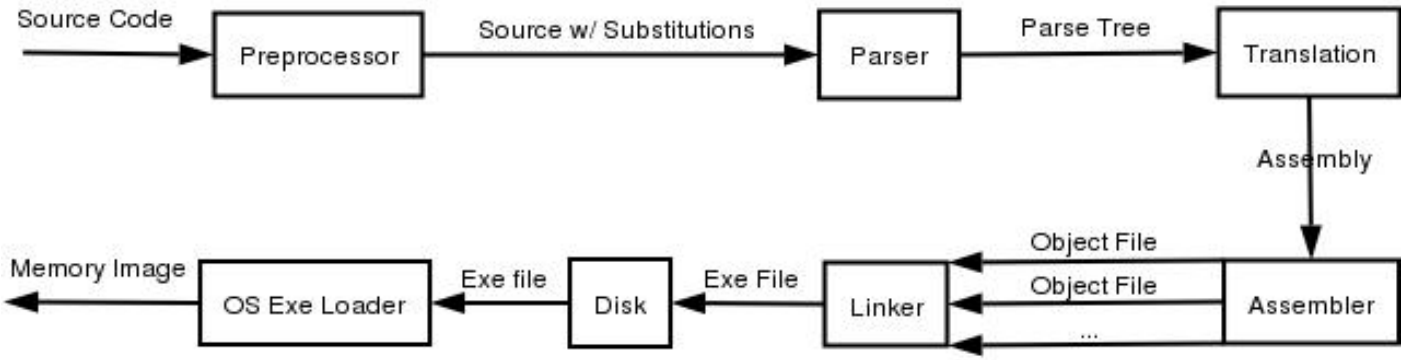

Figure 1. The compilation Process

All 5 stages are implemented by one program in UNIX, namely cc, or in our case, gcc (or g++). The general order of things goes gcc -> gcc -E -> gcc -S -> as -> ld.

Under Microsoft Windows©, however, the process is a bit more obfuscated, but once you delve under the MSVC++ front end, it is essentially the same. Also note that the GNU toolchain is available under Microsoft Windows©, through both the MinGW project as well as the Cygwin Project and behaves the same as under UNIX. Cygwin provides an entire POSIX compatibility layer and UNIX-like environment, where as MinGW just provides the GNU buildchain itself, and allows you to build native windows apps without having to ship an additional dll. Many other commercial compilers exist, but they are omitted for space.

The Compiler

Despite their seemingly disparate approaches to the development environment, both UNIX and Microsoft Windows© do share a common architectural back-end when it comes to compilers (and many many other things, as we will find out in the coming pages). Executable generation is essentially handled end-to-end on both systems by one program: the compiler. Both systems have a single front-end executable that acts as glue for essentially all 5 steps mentioned above.

gcc

gcc is the C compiler of choice for most UNIX. The program gcc itself is actually just a front end that executes various other programs corresponding to each stage in the compilation process. To get it to print out the commands it executes at each step, use `gcc -v`.

cl.exe

cl.exe is the back end to MSVC++, which is the the most prevalent development environment in use on Microsoft Windows©. You'll find it has many options that are quite similar to gcc. Try running `cl -?` for details.

The problem with running cl.exe outside of MSVC++ is that none of your include paths or library paths are set. Running the program `vsvars32.bat` in the CommonX/Tools directory will give you a shell with all the appropriate environment variables set to compile from the command line. If you're a fan of Cygwin, you may find it more comfortable to cut and paste vsvars32.bat into cygwin.bat.

The C Preprocessor

The preprocessor is what handles the logic behind all the # directives in C. It runs in a single pass, and essentially is just a substitution engine.

gcc -E

`gcc -E` runs only the preprocessor stage. This places all include files into your .c file, and also translates all macros into inline C code. You can add `-o file` to redirect to a file.

cl -E

Likewise, `cl -E` will also run only the preprocessor stage, printing out the results to standard out.

Parsing And Translation Stages

The parsing and translation stages are the most useful stages of the compiler. Later in this article, we will use this functionality to teach ourselves assembly, and to get a feel for the type of code generated by the compiler under certain circumstances. Unfortunately, the UNIX world and the Microsoft Windows© world diverge on their choice of syntax for assembly, as we shall see in a bit. It is our hope that exposure to both of these syntax methods will increase the flexibility of the reader when moving between the two environments. Note that most of the GNU tools do allow the flexibility to choose Intel syntax, should you wish to just pick one syntax and stick with it. We will cover both, however.

gcc -S

`gcc -S` will take .c files as input and output .s assembly files in AT&T syntax. If you wish to have Intel syntax, add the option `-masm=intel`. To gain some association between variables and stack usage, use add `-fverbose-asm` to the flags.

gcc can be called with various optimization options that can do interesting things to the assembly code output. There are between 4 and 7 general optimization classes that can be specified with a -ON, where 0 <= N <= 6. 0 is no optimization (default), and 6 is usually maximum, although oftentimes no optimizations are done past 4, depending on architecture and gcc version.

There are also several fine-grained assembly options that are specified with the -f flag. The most interesting are -funroll-loops, -finline-functions, and -fomit-frame-pointer. Loop unrolling means to expand a loop out so that there are n copies of the code for n iterations of the loop (ie no jmp statements to the top of the loop). On modern processors, this optimization is negligible. Inlining functions means to effectively convert all functions in a file to macros, and place copies of their code directly in line in the calling function (like the C++ inline keyword). This only applies for functions called in the same C file as their definition. It is also a relatively small optimization. Omitting the frame pointer (aka the base pointer) frees up an extra register for use in your program. If you have more than 4 heavily used local variables, this may be rather large advantage, otherwise it is just a nuisance (and makes debugging much more difficult).

cl -S

Likewise, cl.exe has a -S option that will generate assembly, and also has several optimization options. Unfortunately, cl does not appear to allow optimizations to be controlled to as fine a level as gcc does. The main optimization options that cl offers are predefined ones for either speed or space. A couple of options that are similar to what gcc offers are:

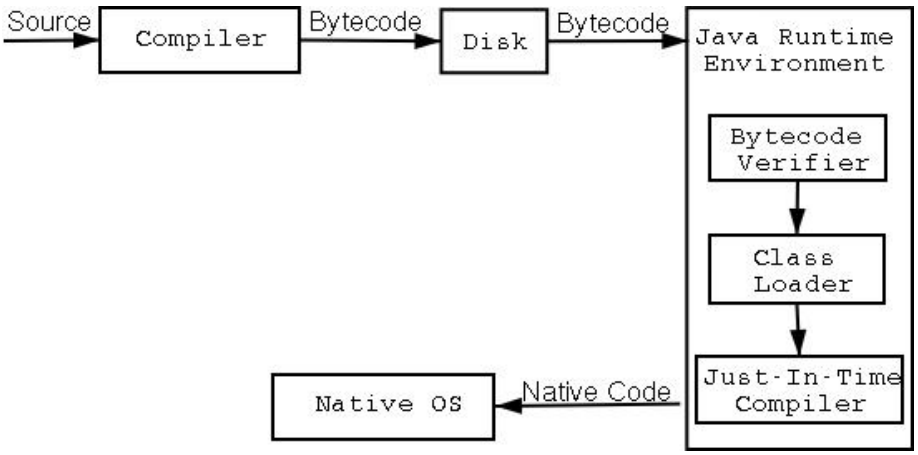

Figure 2. The Java Compile/Execute Path

-Ob<n> – inline functions (-finline-functions)

-Oy – enable frame pointer omission (-fomit-frame-pointer

Assembly Stage

The assembly stage is where assembly code is translated almost directly to machine instructions. Some minimal preprocessing, padding, and instruction reordering can occur, however. We won't concern ourselves with that too much, as it will become visible during disassembly.

GNU as

as is the GNU assembler. It takes input as an AT&T or Intel syntax asm file and generates a .o object file.

MASM

MASM is the Microsoft© assembler. It is executed by running ml.

Linking Stage

Both Microsoft Windows© and UNIX have similar linking procedures, although the support is slightly different. Both systems support 3 styles of linking, and both implement these in remarkably similar ways.

Static Linking

Static linking means that for each function your program calls, the assembly to that function is actually included in the executable file. Function calls are performed by calling the address of this code directly, the same way that functions of your program are called.

Dynamic Linking

Dynamic linking means that the library exists in only one location on the entire system, and the operating system's virtual memory system will map that single location into your program's address space when your program loads. The address at which this map occurs is not always guaranteed, although it will remain constant once the executable has been built. Functions calls are performed by making calls to a compile-time generated section of the executable, called the Procedure Linkage Table, PLT, or jump table, which is essentially a huge array of jump instructions to the proper addresses of the mapped memory.

Runtime Linking

Runtime linking is linking that happens when a program requests a function from a library it was not linked against at compile time. The library is mapped with `dlopen()` under UNIX, and `LoadLibrary()` under Microsoft Windows©, both of which return a handle that is then passed to symbol resolution functions (`dlsym()` and `GetProcAddress()`), which actually return a function pointer that may be called directly from the program as if it were any normal function. This approach is often used by applications to load user-specified plugin libraries with well-defined initialization functions. Such initialization functions typically report further function addresses to the program that loaded them.

ld/collect2

ld is the GNU linker. It will generate a valid executable file. If you link against shared libraries, you will want to actually use what gcc calls, which is collect2.

link.exe

This is the MSVC++ linker. Normally, you will just pass it options indirectly via cl's -link option. However, you can use it directly to link object files and .dll files together into an executable. For some reason though, Microsoft Windows© requires that you have a .lib (or a .def) file in addition to your .dlls in order to link against them. The .lib file is only used in the interim stages, but the location to it must be specified on the -LIBPATH: option.

Java Compilation Process

Java is "semi-interpreted" language and it differs from C/C++ and the process described above. What do we mean by "semi-interpreted" language? Java programs execute in the Java Virtual Machine (or JVM), which makes it an interpreted language. On the other hand Java unlike pure interpreted languages passes through an intermediate compilation step. Java code does not compile to native code that the operating system executes on the CPU, rather the result of Java program compilation is intermediate bytecode. This bytecode runs in the virtual machine. Let us take a look at the process through which the source code is turned into executable code and the execution of it.

Java requires each class to be placed in its own source file, named with the same name as the class name and added suffix `.java`. This basicaly forces any medium sized program to be split in several source files. When compiling source code, each class is placed in its own `.class` file that contains the bytecode. The java compiler differs from gcc/g++ in the fact that if the class you are compiling is dependent on a class that is not compiled or is modified since it was last compiled, it will compile those additional classes for you. It acts similarly to *make*, but is nowhere close to it. After compiling all source files, the result will be at least as much class files as the sources, which will combine to form your Java program. This is where the class loader comes into picture along with the bytecode verifier – two unique steps that distinguish Java from languages like C/C++.

The class loader is responsible for loading each class' bytecode. Java provides developers with the opportunity to write their own class loader, which gives developers great flexibility. One can write a loader that fetches the class from everywhere, even IRC DCC connection. Now let us look at the steps a loader takes to load a class.

When a class is needed by the JVM the `loadClass(String name, boolean resolve);` method is called passing the class name to be loaded. Once it finds the file that contains the bytecode for the class, it is read into memory and passed to the `defineClass`. If the class is not found by the loader, it can delegate the loading to a parent class loader or try to use `findSystemClass` to load the class from local filesystem. The Java Virtual Machine Specification is vague on the subject of when and how the ByteCode verifier is invoked, but by a simple test we can infer that the `defineClass` performs the bytecode verification. (FIXME maybe show the

test). The verifier does four passes over the bytecode to make sure it is safe. After the class is successfully verified, its loading is completed and it is available for use by the runtime.

The nature of the Java bytecode allows people to easily decompile class files to source. In the case where default compilation is performed, even variable and method names are recovered. There are bunch of decompilers out there, but a free one that works well is Jad.

NOW THE FUN BEGINS. THE FIRST STEP IS TO FIGURING OUT WHAT IS GOING ON IN OUR TARGET PROGRAM IS TO GATHER AS MUCH INFORMATION AS WE CAN.SEVERAL TOOLS ALLOW US TO DO THIS ON BOTH PLATFORMS. LET'S TAKE A LOOK AT THEM.

System Wide Process Information

On Microsoft Windows© as on Linux, several applications will give you varying amounts of information about processes running. However, there is a one stop shop for information on both systems.

/proc

The Linux `/proc` filesystem contains all sorts of interesting information, from where libraries and other sections of the code are mapped, to which files and sockets are open where. The `/proc` filesystem contains a directory for each currently running process. So, if you started a process whose pid was 1337, you could enter the directory `/proc/1337/` to find out almost anything about this currently running process. You can only view process information for processes which you own.

The files in this directory change with each UNIX OS. The interesting ones in Linux are: cmdline – lists the command line parameters passed to the process cwd – a link to the current working directory of the process environ – a list of the environment variables for the process exe – the link to the process executable fd – a list of the file descriptors being used by the process maps – VERY USEFUL. Lists the memory locations in use by this process. These can be viewed directly with gdb to find out various useful things.

Sysinternals Process Explorer

Sysinternals provides an all-around must-have set of utilities. In this case, Process Explorer is the functional equivalent of `/proc`. It can show you dll mapping information, right down to which functions are at which addresses, as well as process properties, which includes an environment tab, security attributes, what files and objects are open, what the type of objects those handles are for, etc. It will also allow you to modify processes for which you have access to in ways that are not possible in `/proc`. You can close handles, change permissions, open debug windows, and change process priority.

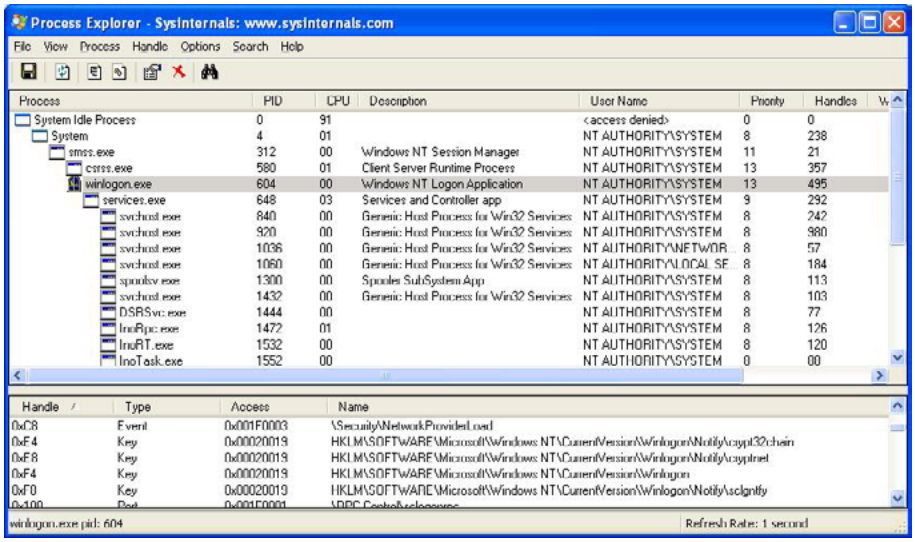

Figure 3. Process Explorer

Obtaining Linking information

The first step towards understanding how a program works is to analyze what libraries it is linked against. This can help us immediately make predictions as to the type of program we're dealing with and make some insights into its behavior.

ldd

ldd is a basic utility that shows us what libraries a program is linked against, or if its statically linked. It also gives us the addresses that these libraries are mapped into the program's execution space, which can be handy for following function calls in disassembled output (which we will get to shortly).

depends

depends is a utility that comes with the Microsoft© SDK, as well as with MS Visual Studio. It will show you quite a bit about the linking information for a program. Not only will list dll's, but it will list which functions in those DLL's are being imported (used) by the current executable, and how they are imported, and then do this recursively for all dll's linked against the executable.

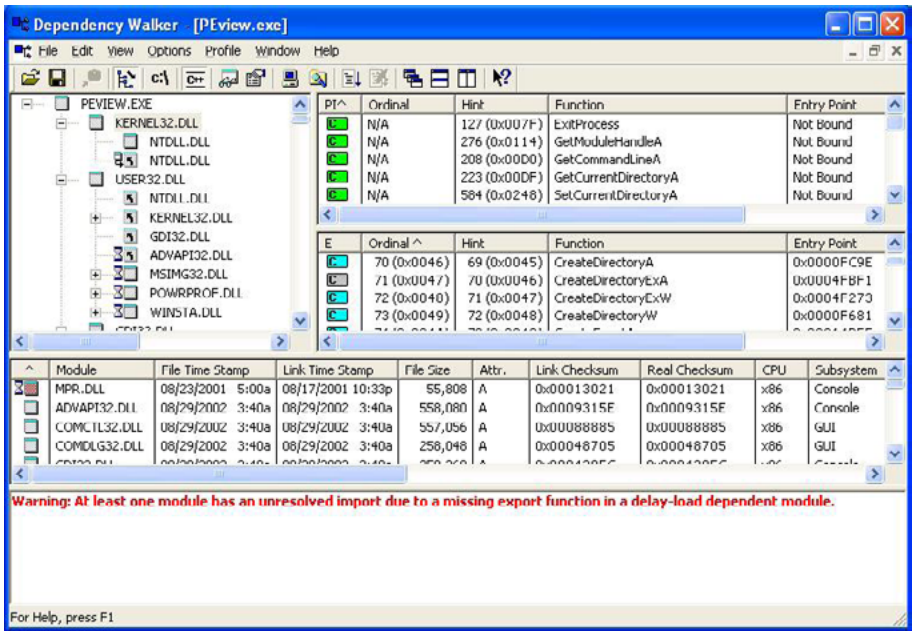

Figure 4. Depends

The layout is a little bit much to process at first. When you click on a DLL, you get the functions from this DLL imported by its parent in the tree (upper right, in green). You also get a list of all the functions that this DLL exports. Those that also present in the imports pane are light blue with a dark blue dot. Those that are called somewhere in the entire linked maze are blue, and those that aren't used at all are grey. Most often all that is used to determine the location of the function is a string and/or an ordinal number, which specifies the numeric index of this function in the export table. Sometimes, the function will be "bound", which means that the linker took a guess at it's location in memory and filled it in. Note that bindings may be rejected as "stale", however, so modifiying this value in the executable won't always give you the results you suspect.

Obtaining Function Information

The next step in reverse engineering is the ability to differentiate functional blocks in programs. Unfortunately, this can prove to be quite difficult if you aren't lucky enough to have debug information enabled. We'll discuss some of those techniques later.

nm

nm lists all of the local and library functions, global variables, and their addresses in the binary. However, it will not work on binaries that have been stripped with strip.

dumpbin.exe

Unfortunately, the closest thing Microsoft Windows© has to nm is dumpbin.exe, which isn't very great. The only thing it can do is essentially what depends already does: that is list functions used by this binary (dumpbin `/imports`), and list functions provided by this binary (dumpbin `/exports`). The only way a binary can export a function (and thus the only way the function is visible) is if that function has the `__declspec(dllexport)` tag next to it's prototype.

Luckily, depends is so overkill, it often provides us with more than the information we need to get the job done. Furthermore, the cygwin port of objdump also gets the job done a lot of the time.

Viewing Filesystem Activity

lsof

lsof is a program that lists all open files by the processes running on a system. An open file may be a regular file, a directory, a block special file, a character special file, an executing text reference, a library, a stream or a network file (Internet socket, NFS file or UNIX domain socket). It has plenty of options, but in its default mode it gives an extensive listing of the opened files. lsof does not come installed by default with most of the flavors of Linux/UNIX, so you may need to install it by yourself. On some distributions lsof installs in `/usr/sbin` which by default is not in your path and you will have to add it. An example output would be:

```
COMMAND     PID  USER   FD   TYPE     DEVICE     SIZE       NODE NAME
bash        101 nasko  cwd    DIR        3,2     4096    1172699 /home/nasko
bash        101 nasko  rtd    DIR        3,2     4096          2 /
bash        101 nasko  txt    REG        3,2   518140    1204132 /bin/bash
bash        101 nasko  mem    REG        3,2   432647     748736 /lib/ld-2.2.3.so
bash        101 nasko  mem    REG        3,2    14831    1399832 /lib/libtermcap.so.2.0.8
bash        101 nasko  mem    REG        3,2    72701     748743 /lib/libdl-2.2.3.so
bash        101 nasko  mem    REG        3,2  4783716     748741 /lib/libc-2.2.3.so
bash        101 nasko  mem    REG        3,2   249120     748742 /lib/libnss_compat-2.2.3.so
bash        101 nasko  mem    REG        3,2   357644     748746 /lib/libnsl-2.2.3.so
bash        101 nasko    0u   CHR        4,5              260596 /dev/tty5
bash        101 nasko    1u   CHR        4,5              260596 /dev/tty5
bash        101 nasko    2u   CHR        4,5              260596 /dev/tty5
bash        101 nasko  255u   CHR        4,5              260596 /dev/tty5
screen      379 nasko  cwd    DIR        3,2     4096    1172699 /home/nasko
screen      379 nasko  rtd    DIR        3,2     4096          2 /
screen      379 nasko  txt    REG        3,2   250336     358394 /usr/bin/screen-3.9.9
screen      379 nasko  mem    REG        3,2   432647     748736 /lib/ld-2.2.3.so
screen      379 nasko  mem    REG        3,2   357644     748746 /lib/libnsl-2.2.3.so
screen      379 nasko    0r   CHR        1,3              260468 /dev/null
screen      379 nasko    1w   CHR        1,3              260468 /dev/null
screen      379 nasko    2w   CHR        1,3              260468 /dev/null
screen      379 nasko    3r  FIFO        3,2             1334324 /home/nasko/.screen/379.pts-6.
slack
startx      729 nasko  cwd    DIR        3,2     4096    1172699 /home/nasko
startx      729 nasko  rtd    DIR        3,2     4096          2 /
startx      729 nasko  txt    REG        3,2   518140    1204132 /bin/bash
ksmserver   794 nasko    3u  unix 0xc8d36580              346900 socket
ksmserver   794 nasko    4r  FIFO        0,6              346902 pipe
```

```
ksmserver   794 nasko    5w  FIFO        0,6              346902 pipe
ksmserver   794 nasko    6u  unix 0xd4c83200              346903 socket
ksmserver   794 nasko    7u  unix 0xd4c83540              346905 /tmp/.ICE-unix/794
mozilla-b  5594 nasko  144u  sock        0,0              639105 can't identify protocol
mozilla-b  5594 nasko  146u  unix 0xd18ec3e0              639134 socket
mozilla-b  5594 nasko  147u  sock        0,0              639135 can't identify protocol
mozilla-b  5594 nasko  150u  unix 0xd18ed420              639151 socket
```

Here is brief explanation of some of the abbreviations lsof uses in its output:

```
cwd  current working directory
mem  memory-mapped file
pd   parent directory
rtd  root directory
txt  program text (code and data)
CHR  for a character special file
sock for a socket of unknown domain
unix for a UNIX domain socket
DIR  for a directory
FIFO for a FIFO special file
```

It is pretty handy tool when it comes to investigating program behavior. lsof reveals plenty of information about what the process is doing under the surface.

Fuser
A command closely related to lsof is fuser. fuser accepts as a command-line parameter the name of a file or socket. It will return the pid of the process accessing that file or socket.

Sysinternals Filemon

The analog to lsof in the windows world is the Sysinternals Filemon utility. It can show not only open files, but reads, writes, and status requests as well. Furthermore, you can filter by specific process and operation type. A very useful tool. (FIXME: This has a Linux version as well).

Sysinternals Regmon

The registry in Microsoft Windows© is a key part of the system that contains lots of secrets. In order to try and understand how a program works, one definitely should know how the target interacts with the registry. Does it store configuration information, passwords, any useful information, and so on. Regmon from Sysinternals lets you monitor all or selected registry activity in real time. Definitely a must if you plan to work on any target on Microsoft Windows©.

Viewing Open Network Connections

So this is one of the cases where both Linux and Microsoft Windows© have the same exact name for a utility, and it performs the same exact duty. This utility is netstat.

netstat

netstat is handy little tool that is present on all modern operating systems. It is used to display network connections, routing tables, interface statistics, and more.

How can netstat be useful? Let's say we are trying to reverse engineer a program that uses some network communication. A quick look at what netstat displays can give us clues where the program connects and after some investigation maybe why it connects to this host. netstat does not only show TCP/IP connections, but also UNIX domain socket connections which are used in interprocess communication in lots of programs. Here is an example output of it:

Listing 1. Netstat output

```
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 slack.localnet:58705    egon:ssh                ESTABLISHED
tcp        0      0 slack.localnet:51766    gw.localnet:ssh         ESTABLISHED
tcp        0      0 slack.localnet:51765    gw.localnet:ssh         ESTABLISHED
tcp        0      0 slack.localnet:38980    clortho:ssh             ESTABLISHED
tcp        0      0 slack.localnet:58510    students:ssh            ESTABLISHED
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node Path
unix  5      [ ]         DGRAM                    68     /dev/log
unix  3      [ ]         STREAM     CONNECTED     572608 /tmp/.ICE-unix/794
unix  3      [ ]         STREAM     CONNECTED     572607
unix  3      [ ]         STREAM     CONNECTED     572604 /tmp/.X11-unix/X0
unix  3      [ ]         STREAM     CONNECTED     572603
unix  2      [ ]         STREAM                   572488
```

NOTE
The output shown is from Linux system. The Microsoft Windows© output is almost identical.

As you can see there is great deal of info shown by netstat. But what is the meaning of it? The output is divided in two parts – Internet connections and UNIX domain sockets as mentioned above. Here is breifly what the Internet portion of netstat output means. The first column shows the protocol being used (tcp, udp, unix) in the particular connection. Receiving and sending queues for it are displayed in the next two columns, followed by the information identifying the connection – source host and port, destination host and port. The last column of the output shows the state of the connection. Since there are several stages in opening and closing TCP connections, this field was included to show if the connection is ESTABLISHED or in some of the other available states. SYN_SENT, TIME_WAIT, LISTEN are the most often seen ones. To see complete list of the available states look in the man page for netstat. FIXME: Describe these states.

Depending on the options being passed to netstat, it is possible to display more info. In particular interesting for us is the -p option (not available on all UNIX systems). This will show us the program that uses the connection shown, which may help us determine the behaviour of our target. Another use of this options is in tracking down spyware programs that may be installed on your system. Showing all the network connection and looking for unknown entries is invaluable tool in discovering programs that you are unaware of that send information to the network. This can be combined with the -a option to show all connections. By default listening sockets are not displayed in netstat. Using the -a we force all to be shown. -n shows numerical IP addesses instead of hostnames.

```
netstat -p as normal user
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 slack.localnet:58705    egon:ssh                ESTABLISHED -
tcp        0      0 slack.localnet:58766    winston:www             ESTABLISHED 5587/mozilla-bin


netstat -npa as root user
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:139             0.0.0.0:*               LISTEN      390/smbd
tcp        0      0 0.0.0.0:6000            0.0.0.0:*               LISTEN      737/X
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      78/sshd
tcp        0      0 10.0.0.3:58705          128.174.252.100:22      ESTABLISHED 13761/ssh
tcp        0      0 10.0.0.3:51766          10.0.0.1:22             ESTABLISHED 897/ssh
tcp        0      0 10.0.0.3:51765          10.0.0.1:22             ESTABLISHED 896/ssh
```

```
tcp        0      0 10.0.0.3:38980          128.174.252.105:22      ESTABLISHED 8272/ssh
tcp        0      0 10.0.0.3:58510          128.174.5.39:22         ESTABLISHED 13716/ssh
```

So this output shows that mozilla has established a connection with winston for HTTP traffic (since port is www(80)). In the second output we see that the SMB daemon, X server, and ssh daemon listen for incoming connections.

Gathering Network Data

Collecting network data is usually done with a program called sniffer. What the program does is to put your ethernet card into promiscuous mode and gather all the information that it sees. What is a promiscuous mode? Ethernet is a broadcast media. All computers broadcast their messages on the wire and anyone can see those messages. Each network interface card (NIC), as a hardcoded physical address called MAC (Media Access Control) address, which is used in the Ethernet protocol. When sending data over the wire, the OS specifies the destination of the data and only the NIC with the destination MAC address will actually process the data. All other NICs will disregard the data coming on the wire. When in promiscuous mode, the card picks up all the data that it sees and sends it to the OS. In this case you can see all the data that is flowing on your local network segment.

Disclaimer
Switched networks eliminate the broadcast to all machines, but sniffing traffic is still possible using certain techniques like ARP poisoning. (FIXME: link with section on ARP poisoning if we have one.)

Several popular sniffing programs exist, which differ in user interface and capabilities, but any one of them will do the job. Here are some good tools that we use on a daily basis:

- ethereal – one of the best sniffers out there. It has a graphical interface built with the GTK library. It is not just a sniffer, but also a protocol analyzer. It breaks down the captured data into pieces, showing the meaning of each piece (for example TCP flags like SYN or ACK, or even kerberos or NTLM headers). Furthermore, it has excellent packet filtering mechanisms, and can save captures of network traffic that match a filter for later analysis. It is available for both Microsoft Windows© and Linux and requires (as almost any sniffer) the pcap library. Ethereal is available at *www.ethereal.com* and you will need libpcap for Linux orWinPcap for Microsoft Windows©.

- tcpdump – one of the first sniffing programs. It is a console application that prints info to the screen. The advantage is that it comes by default with most Linux distributions. Microsoft Windows© version is available as well, called WinDump.

- ettercap – also a console based sniffer. Uses the ncurses library to provide console GUI. It has built in ARP poisoning capability and supports plugins, which give you the power to modify data on the fly. This makes it very suitable for all kinds of Man-In-The-Middle attacks (MITM), which will we will describe in chapter (FIXME: link). Ettercap isn't that great a sniffer, but nothing prevents you from using its ARP poisoning and plugin features while also running a more powerful sniffer such as ethereal.

Now that you know what a sniffer is and hopefully learned how to use basic functionality of your favorite one, you are all set to gather network data. Let's say you want to know how does a mail client authenticate and fetch messages from the server. Since the protocol in use is POP3, we should instruct ethereal (our sniffer of choice) to capture traffic only destined to port 110 or originating from port 110. In our case since we want to capture both directions of the traffic we can set the filter to be `tcp.port == 110`. If you have a lot of machines checking mail at the same time on a network with a hub, you might want to restrict the matching only to your machine and the server you are connecting to. Here is an example of captured packet in ethereal:

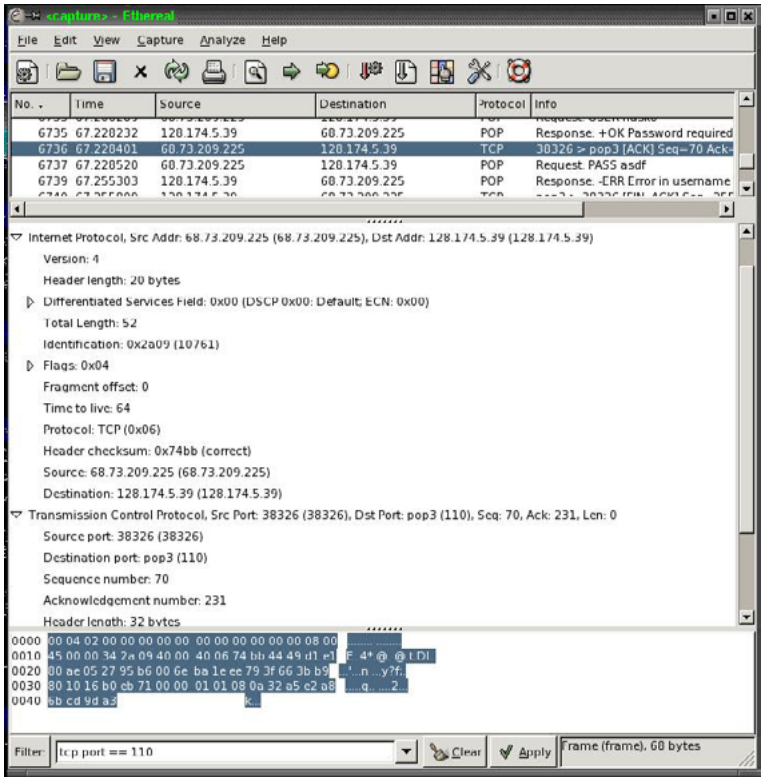

Figure 5. Ethereal capture

Ethereal breaks down the packet for us, showing what each part of the data means. For example, it shows that the Internet Protocol version is 4 or that the header checksum is 0x74bb and is in fact the correct checksum for that packet. It shows in similar manner details for each part of the header and the data at the end of the packet if any.

Using packet captures, one can trace the flow of a protocol to better understand how an application works, or even try to reverse engineer the protocol itself if unknown.

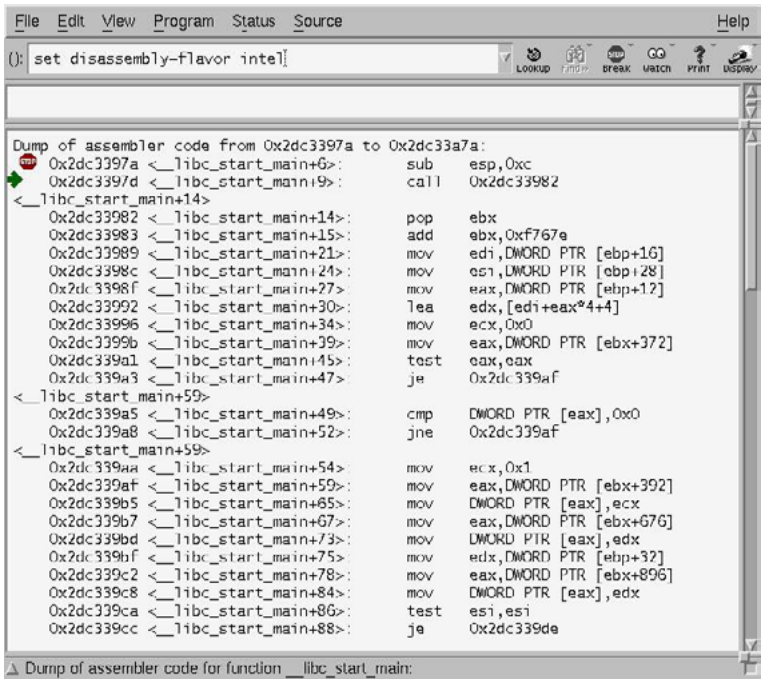

Figure 6. ASM in DDD

Determining Program Behavior

There are a couple of tools that allow us to look into program behavior at a more closer level. Lets look at some of these:

Tracing System Calls

This section is really only relevant for to our efforts under UNIX, as Microsoft Windows© system calls change regularly from version to version, and have unpredictable entry points.

strace/truss(Solaris)

These programs trace system calls a program makes as it makes them.

Useful options:

-f (follow fork)

-ffo filename (output trace to filename.pid for forking)

-i (Print instruction pointer for each system call)

Tracing Library Calls

Now we're starting to get to the more interesting stuff. Tracing library calls is a very powerful method of system analysis. It can give us a *lot* of information about our target.

ltrace

This utility is extremely useful. It traces ALL library calls made by a program.

Useful options:

-S (display syscalls too)

-f (follow fork)

-o filename (output trace to filename)

-C (demangle C++ function call names)

-n 2 (indent each nested call 2 spaces)

-i (prints instruction pointer of caller)

-p pid (attaches to specified pid)

API Monitor

API Monitor is incredible. It will let you watch .dll calls in real time, filter on type of dll call, view.

HERE I AM SKIPPING A FEW THINGS, CAUSE I DON'T CONSIDER THEM TO BE IMPORTANT PLUS THIS WILL ONLY LENGTHEN THE ARTICLE

User-level Debugging

DDD

DDD is the Data Display Debugger, and is a nice GUI front-end to gdb, the GNU debugger. For a long time, the authors believed that the only thing you really needed to debug was gdb at the command line. However, when reverse engineering, the ability to keep multiple windows open with stack contents, register values, and disassembly all on the same workspace is just too valuable to pass up.

Also, DDD provides you with a gdb command line window, and so you really aren't missing anything by using it. Knowing gdb commands is useful for doing things that the UI is too clumsy to do quickly. gdb has a nice built-in help system organized by topic. Typing help will show you the categories. Also, DDD will update the gdb window with commands that you select from the GUI, enabling you to use the GUI to help you learn the gdb command line. The main commands we will be interested in are run, break, cont, stepi, nexti, finish, disassemble, bt, info [registers/frame], and x. Every command in gdb can be followed by a number N, which means repeat N times. For example, stepi 1000 will step over 1000 assembly instructions.

Setting Breakpoints

A breakpoint stops execution at a particular location. Breakpoints are set with the break command, which can take a function name, a filename:line_number, or *0xaddress. For example, to set a breakpoint at the aforementioned `__libc_start_main()`, simply specify `break __libc_start_main`. In fact, gdb even has tab completion, which will allow you to tab through all the symbols that start with a particular string (which, if you are dealing with a production binary, sadly won't be many).

Viewing Assembly

Ok, so now that we've got a breakpoint set somewhere, (let's say `__libc_start_main`). To view the assembly in DDD, go to the View menu and select source window. As soon as we enter a function, the disassembly will be shown in the bottom half of the source window. To change the syntax to the more familar Intel variety, go to Edit->Gdb Settings... under Disassembly flavor. This can also be accomplished with `set disassembly-flavor intel` from the gdb prompt. But using the DDD menus will save your settings for future sessions.

Viewing Memory and the Stack

In gdb, we can easily view the stack by using the x command. x stands for Examine Memory, and takes the syntax `x /<Number><format letter><size letter> <ADDRESS>`. Format letters are (octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char) and s(string). Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes). For example, `x /32xw 0x400000` will dump 32 words (32 bit integers) starting at 0x400000. Note that you can also use registers in place of the address, if you prefix them with a $. For example, `x /32xw $esp` will view the top 32 words on the stack.

DDD has some nice capabilities for viewing arbitrary dumps of memory relating to the registers. Go to View->Data Window... Once the Data Window is open, go to Display (hold down the mouse button as you click), and go to Other.. You can type in any symbol, variable, expression, or gdb command (in backticks) in this window, and it will be updated every time you issue a command to the debugger. A couple good ones to do would be `x /32xw $esp` and `x/16sb $esp`. Click the little radio button to add these to the menu, and you can then open the stack from this display and it will be updated in real time as you step through your program.

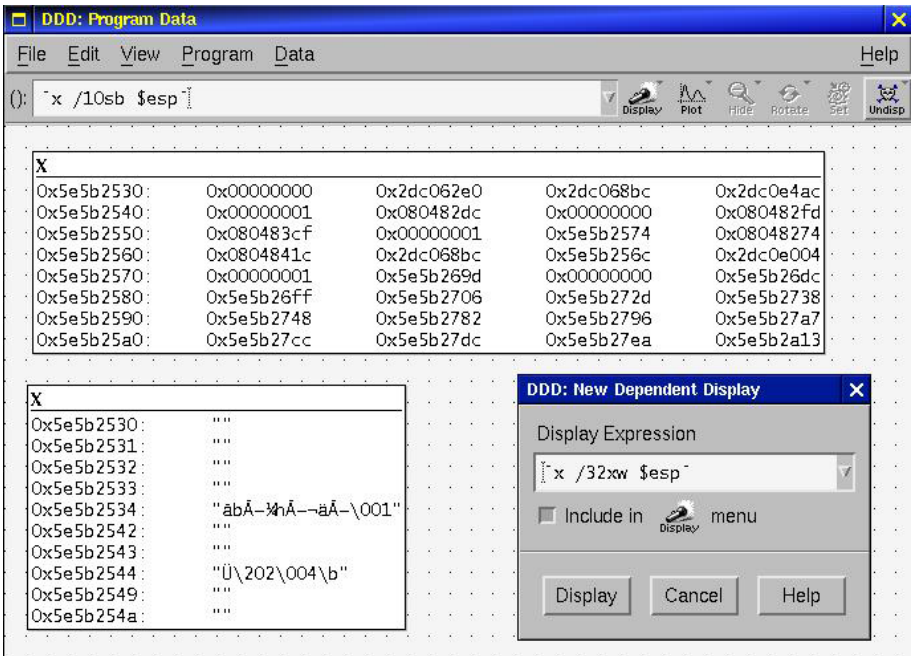

Figure 7. Stack Displays with New Display Window

Viewing Memory as Specific Data Structures

So DDD has fantastic ability to lay out data structures graphically, also trough the Display window mentioned above. Simply cast a memory address to a pointer of a particular type, and DDD will plot the structure in its graph window. If the data structure contains any pointers, you can click on these and DDD will open up a display for that structure as well.

Oftentimes, programs we're interested in won't have any debugging symbols, and as such, we won't be able to view any structures in an easy to understand form. For seldom used structures, this isn't that big of a deal, as you can just take them apart using the `x` command. However, if you are dealing with more complicated data structures, you may want to have a set of types available to use again and again. Luckily, through the magic of the ELF format, this is relatively easy to achieve. Simply define whatever structures or classes you suspect are used and include whatever headers you require in a .c file, and then compile it with `gcc -shared`. This will produce a .so file. Then, from within gdb but before you begin debugging, run the command `set env LD_PRELOAD=file.so`. From then on, you will be able to use these types in that gdb/DDD session as if they were compiled in to the program itself. (FIXME: Come up with a good example for this).

Using watchpoints

-> Example using gdb to set breakpoints in functions with and without debugging symbols.

-> FIXME: Test watchpoints

WinDbg

WinDbg is part of the standart Debugging Tools for Microsoft Windows© that everyone can download for free from. Microsoft© offers few different debuggers, which use common commands for most operations and ofcourse there are cases where they differ. Since WinDbg is a GUI program, all operations are supposed to be done using the provided visual components. There is also a command line embeded in the debugger, which lets you type commands just like if you were to use a console debugger like `ntsd`. The following section briefly mentions what commands are used to do common everyday tasks. For more complete documentation check the Help file that comes with WinDbg. An example debugging session is presented to help clarify the usage of the most common commands.

Breakpoints

Breakpoints can be set, unset, or listed with the GUI by using Edit->Breakpoints or the shortcut keys Alt+F9. From the command line one can set breakpoints using the `bp` command, list them using `bl` command, and delete them using `bc` command. One can set breakpoints both on function names (provided the symbol files are available) or on a memory address. Also if source file is available the debugger will let you set breakpoints on specific lines using the format `bX filename:linenumber`.

Viewing Assembly

In WinDbg you can use View->Disassembly option to open a window which will show you the disassembly of the current context. In `ntsd` you can use the `u` to view the disassembled code.

Stack operations

There are couple of things one usually does with the stack. One is to view the frames on the stack, so it can be determined which function called which one and what is the current context. This is done using the `k` command and its variations. The other common operation is to view the elements on the stack that are part of the current stack frame. The easiest way to do so is using `db esp ebp`, but it has its limitations. It assumes that the `%ebp` register actually points to the begining of the stack frame. This is not always true, since omission of the frame pointer is common optimization technique. If this is the case, you can always see what the `%esp` register is pointing to and start examining memory from that address. The debugger also allows you to "walk" the stack. You can move to any stack frame using `.frame X` where `X` is the number of the frame. You can easily get the frame numbers using `kn`. Keep in mind that the frames are counted starting from 0 at the frame on top of the stack.

Reading and Writing to Memory

Reading memory is accomplished with the `d*` commands. Depending on how you want to view the data you use a specific variation of this command. For example to see the address to which a pointer is pointing, we can use `dp` or to view the value of a word, one can use `dw`. The help file says that one can view memory using ranges, but one can also use lengths to make it easy to display memory. For example if we want to see 0x10 bytes at memory location 0x77f75a58 you can either say `db 77f75a58 77f75a58+10` or less typing gives you `db 77f75a58 l 10`.

Provided that you have symbols/source files, the `dt` is very helpful. It tries to find the data type of the sybol or memory location and display it accordingly.

Tips and tricks

Knowing your debugger can save you lots of time and pain in debugging either your own programs or when reverse engineering other's. Here are few things we find useful and time saving. This is not a complete list at all. If you know other tricks and want to contribute, let us know. `poi()` – this command dereferences a pointer to give you the value that it is pointing to. Using this with user-defined aliases gives you convinient way of viewing data.

Example

Let's set a breakpoint in on the function main

```
0:000> bp main
*** WARNING: Unable to verify checksum for test.exe
```

Let's set a breakpoint in on the function main

```
0:000> g
Breakpoint 0 hit
eax=003212e8 ebx=7ffdf000 ecx=00000001 edx=7ffe0304 esi=00000a28 edi=00000000
eip=00401010 esp=0012fee8 ebp=0012ffc0 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
test!main:
00401010 55              push    ebp
```

Enable loading of line information if available

```
0:000> .lines
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdll.dll -
Line number information will be loaded
```

Set the stepping to be by source lines

```
0:000> l+t
Source options are 1:
     1/t - Step/trace by source line
```

Enable displaying of source line

```
0:000> l+s
Source options are 5:
     1/t - Step/trace by source line
     4/s - List source code at prompt
```

Start stepping through the program

```
0:000> p
*** WARNING: Unable to verify checksum for test.exe
eax=003212e8 ebx=7ffdf000 ecx=00000001 edx=7ffe0304 esi=00000a28 edi=00000000
eip=00401016 esp=0012fed4 ebp=0012fee4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
>    6:   char array [] = { 'r', 'e', 'v', 'e', 'n', 'g' };
test!main+6:
00401016 c645f072         mov    byte ptr [ebp-0x10],0x72 ss:0023:0012fed4=05
0:000>
eax=003212e8 ebx=7ffdf000 ecx=00000001 edx=7ffe0304 esi=00000a28 edi=00000000
eip=0040102e esp=0012fed4 ebp=0012fee4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
>    7:   int intval = 123456;
test!main+1e:
0040102e c745fc40e20100 mov dword ptr [ebp-0x4],0x1e240 ss:0023:0012fee0=0012ffc0
0:000>
eax=003212e8 ebx=7ffdf000 ecx=00000001 edx=7ffe0304 esi=00000a28 edi=00000000
eip=00401035 esp=0012fed4 ebp=0012fee4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
>    9:   test = (char*) malloc(strlen("Test")+1);
test!main+25:
00401035 6840cb4000       push    0x40cb40
0:000>
eax=00321018 ebx=7ffdf000 ecx=00000000 edx=00000005 esi=00000a28 edi=00000000
eip=00401051 esp=0012fed4 ebp=0012fee4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
>   10:   if (test == NULL) {
test!main+41:
00401051 837df800       cmp dword ptr [ebp-0x8],0x0 ss:0023:0012fedc=00321018
0:000>
eax=00321018 ebx=7ffdf000 ecx=00000000 edx=00000005 esi=00000a28 edi=00000000
eip=00401061 esp=0012fed4 ebp=0012fee4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
>   13:   strncpy(test, "Test", strlen("Test"));
test!main+51:
00401061 6848cb4000       push    0x40cb48
0:000>
eax=00321018 ebx=7ffdf000 ecx=00000000 edx=74736554 esi=00000a28 edi=00000000
```

```
eip=00401080 esp=0012fed4 ebp=0012fee4 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000216
>   14:   test[4] = 0x00;
test!main+70:
00401080 8b4df8          mov     ecx,[ebp-0x8]     ss:0023:0012fedc=00321018
0:000>
eax=00321018 ebx=7ffdf000 ecx=00321018 edx=74736554 esi=00000a28 edi=00000000
eip=00401087 esp=0012fed4 ebp=0012fee4 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000216
>   16:   printf("Hello RevEng-er, this is %s\n", test);
test!main+77:
00401087 8b55f8          mov     edx,[ebp-0x8]     ss:0023:0012fedc=00321018
```

Display the array as bytes and ascii

```
0:000> db array array+5
0012fed4  72 65 76 65 6e 67                                reveng
```

View the type and value of intval

```
0:000> dt intval
Local var @ 0x12fee0 Type int
123456
```

View the type and value of test

```
0:000> dt test
Local var @ 0x12fedc Type char*
0x00321018 "Test"
```

View the memory test points to manually

```
0:000> db 00321018 00321018+4
00321018  54 65 73 74 00                                   Test.
```

Quit the debugger

```
0:000> q
quit:
Unloading dbghelp extension DLL
Unloading exts extension DLL
Unloading ntsdexts extension DLL
```

THINGS I'D BE LEAVING IN THIS ARTICLE

- Executable formats
- Code Modification
- Network Application Interception

LOOK OUT FOR 2ND ARTICLE OF THIS SERIES FOR ALL THESE.

About the Author

Talking of my education I m just a school going student. I have been working as a penetration tester for more than 2 years now. Have reported many bugs to websites like AVG, The Times Of India, Cartoon Network etc. AVG recognized me by giving a SMIME.P7S certificate. Recently I made my website www.ethicalhackx.com live I also provide hosting services at www.ethicalhostx.com.

AnDevCon

May 27-30, 2014

Sheraton Boston

Register Early and SAVE!

Get the best real-world Android developer training anywhere!

- Choose from more than 75 classes and in-depth tutorials
- Network with speakers and other Android developers
- Check out more than 40 exhibiting companies

Take your Android development skills to the next level!

Find out why you should go to AnDevCon! Watch the videos at www.AnDevCon.com





Register Early and Save at www.AnDevCon.com

A BZ Media Event #AnDevCon

Detecting and Exploiting the OpenSSL-Heartbleed Vulnerability

by Daniel Dieterle

In this article we will discuss how to detect systems that are vulnerable to the OpenSSL-Heartbleed vulnerability and learn how to exploit them using Metasploit on Kali Linux.

The internet has been plastered with news about the OpenSSL heartbeat or "Heartbleed" vulnerability (CVE-2014-0160) that some have said could affect up to 2/3 of the internet. Everything from servers to routers to smart phones could be tricked into giving up encrypted data in plain text.

Basically, OpenSSL is an encryption library used in HTTPS communication. When most think of HTTPS communication, they think of the little lock icon that shows up in the browser bar when you visit online stores and banking websites.

HTTPS is supposed to be the "secured" version of regular HTTP communication. Any data communication using SSL should be secure and encrypted. But with the latest OpenSSL vulnerability, unencrypted information can be recovered from a vulnerable system from specially crafted heartbeat messages.

During communication, OpenSSL uses a "heartbeat" message that echoes back data to verify that it was received correctly. The problem is, in OpenSSL 1.0.1 to 1.0.1f, a hacker can trick OpenSSL by sending a single byte of information but telling the server that it sent up to 64K bytes of data that needs to be checked and echoed back.

And the server will respond with random data – *from its memory*!

During testing, security researchers using this technique have been able to trick vulnerable systems into giving up login credentials, session cookies, and other sensitive data.

But as mentioned the vulnerability goes past just visiting shopping webpages – major websites, social media & game servers, even Android systems, embedded devices and routers could all be affected. Any device that uses the vulnerable version of OpenSSL is at risk.

The vulnerability is remedied in the latest update of OpenSSL, but the problem is it could take years for all the affected devices to be found and patched. And some embedded and proprietary devices may never be patched!

In this article we will quickly discuss some ways to detect the Heartbleed vulnerability, take a deeper look at finding vulnerable systems with the ever popular scanning tool, "Nmap", and finally see how to exploit a vulnerable system using the Heartbleed exploit module in Metasploit.

Before we go any further, it is *neither legal nor ethical* to access servers that you do not own. Never run security scans or checks on systems that you do not own or have approval to scan.

Software used in this Article

For this tutorial I will be using a WordPress server and Kali Linux system running on a Windows 7 system in VMWare Player virtual machines (VMs).

For a vulnerable server, I used one of Turnkey Linux WordPress VMs [1]. For the record, there are security updates available for Turnkey's WordPress, but during the setup, and for this tutorial, I purposefully told this VM *not to install the security updates* so I could test for the OpenSSL vulnerability.

I also downloaded the latest Kali Linux VMware image (1.06) from Offensive Security [2].

Simply boot up both images in your VMWare Player or Workstation. Once the WordPress VM is configured (just answer a few simple questions) you are pretty much set to go. Remember during configuration to choose not to install the security updates automatically, or this tutorial will not work.

Please note that we are basically putting a purposefully vulnerable VM on a computer. Never place a vulnerable VM on a mission critical system or a computer that has open or un-firewalled internet access.

Programs that will Detect Vulnerable Systems

Sadly for those who may not know they have at-risk systems, it is not hard to detect vulnerable systems. Shortly after the vulnerability was publicly announced, a plethora of tools and utilities to check for exploitable systems popped up all over the web.

Unbelievably there are even online lists of the top 10,000 websites that were vulnerable in early April (many have been patched since).

A quick Google search will reveal that there are various utilities, scripts available in several languages, and browser plug-ins (even a honeypot or two) that will detect the bug. I have just never felt comfortable using random "security" tools from the web. Without going through them line by line, you never know what you are getting – *so use them at your own risk*.

Nessus [3] and other main line security programs have updated their scanning engines earlier in the month to detect Heartbleed. If you are a corporate IT center, and haven't done so already, check with your security scanning tool providers to see if they can detect it.

Lastly, and probably the most unnerving for those who haven't patched their systems, you can find vulnerable devices worldwide very quickly using keyword searches in Shodan [4] – the "hacker's Google".

But as a security specialist or business owner, how can you scan your systems if you don't have a capable security program or don't want to trust a random script off the web?

How about using the ever popular scanning tool Nmap?

Detecting Exploit with Nmap

Nmap has created a Heartbleed script that does a great job of detecting vulnerable servers. If the Open-Heartbleed script is not already included in your nmap install, you will need to manually install it.

This is pretty easy, just visit the OpenSSL-Heartbleed nmap Script webpage [5], copy and save the nmap nse script file [6] to your nmap "scripts" directory as seen below:

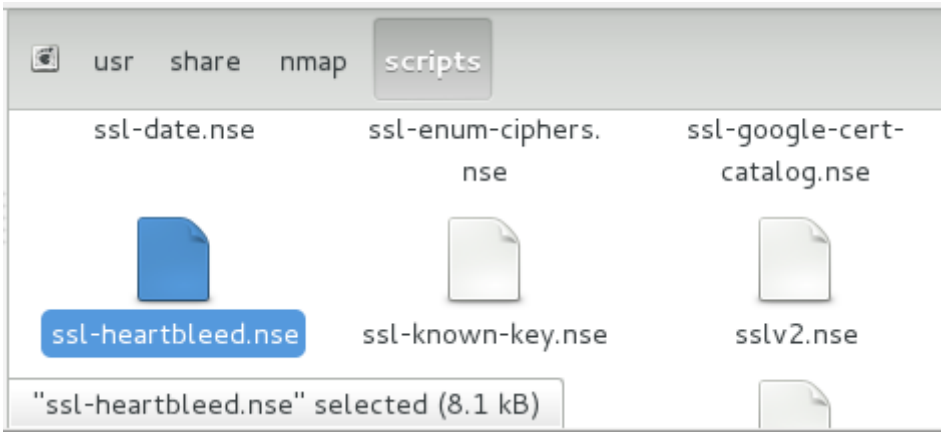

Figure 1. Copying "nmap.nse" file into scripts directory

You will also need the nmap “tls.lua” library file [7], save this to the nmap “nselib” directory as seen below:

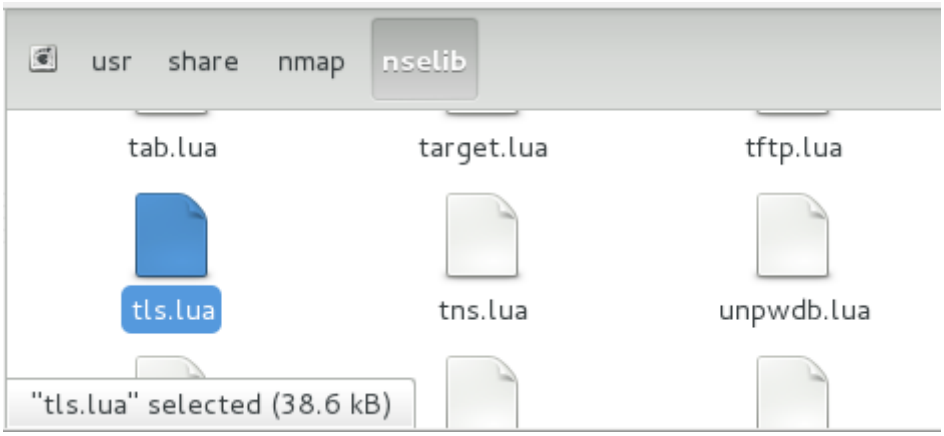

Figure 2. Copying “tls.lua” file into the nselib directory

That’s it; we can now use the heartbleed script in nmap to detect vulnerable systems.

To use the command, the syntax is:

```
nmap -sV --script=ssl-heartbleed <target>
```

All we need to add is the IP address of our test target WordPress site, 192.168.1.70 in this instance:

```
root@kali:/usr/share/nmap# nmap -sV --script=ssl-heartbleed 192.168.1.70
```

Figure 3. Nmap command to scan for Heartbleed vulnerability

And if the target machine is vulnerable we will see this (Figure 4):

```
Starting Nmap 6.40 ( http://nmap.org ) at 2014-04-20 17:49 EDT
Nmap scan report for 192.168.1.70
Host is up (0.00020s latency).
Not shown: 997 closed ports
PORT    STATE SERVICE  VERSION
22/tcp  open  ssh      OpenSSH 6.0p1 Debian 4 (protocol 2.0)
80/tcp  open  http     Apache httpd 2.2.22 ((Debian))
443/tcp open  ssl/http Apache httpd 2.2.22 ((Debian))
| ssl-heartbleed:
|   VULNERABLE:
|   The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptog
raphic software library. It allows for stealing information intended to be prote
cted by SSL/TLS encryption.
|     State: VULNERABLE
|     Risk factor: High
|     Description:
|       OpenSSL versions 1.0.1 and 1.0.2-beta releases (including 1.0.1f and 1.0
.2-beta1) of OpenSSL are affected by the Heartbleed bug. The bug allows for read
ing memory of systems protected by the vulnerable OpenSSL versions and could all
ow for disclosure of otherwise encrypted confidential information as well as the
 encryption keys themselves.
|
|     References:
|       http://www.openssl.org/news/secadv_20140407.txt
|       https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160
|_      http://cvedetails.com/cve/2014-0160/
MAC Address: 00:0C:29:37:4A:26 (VMware)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

Figure 4. Screenshot of Nmap command running

```
State: VULNERABLE
Risk Factor: High
```

As you can see, Nmap scanned our test server and found that it indeed contains a vulnerable version of OpenSSL.

Exploiting with Metasploit

Metasploit has released a couple modules to its framework to deal with the new OpenSSL bug – A server module to test client software and a scanner module.

Now that we know we have a vulnerable server, we can use the Metasploit OpenSSL-Heartbleed scanner module to exploit it. (Note: you can use the module to detect vulnerable systems also)

- Update Metasploit to get the latest modules. Just type “msfupdate” at a Kali command prompt:

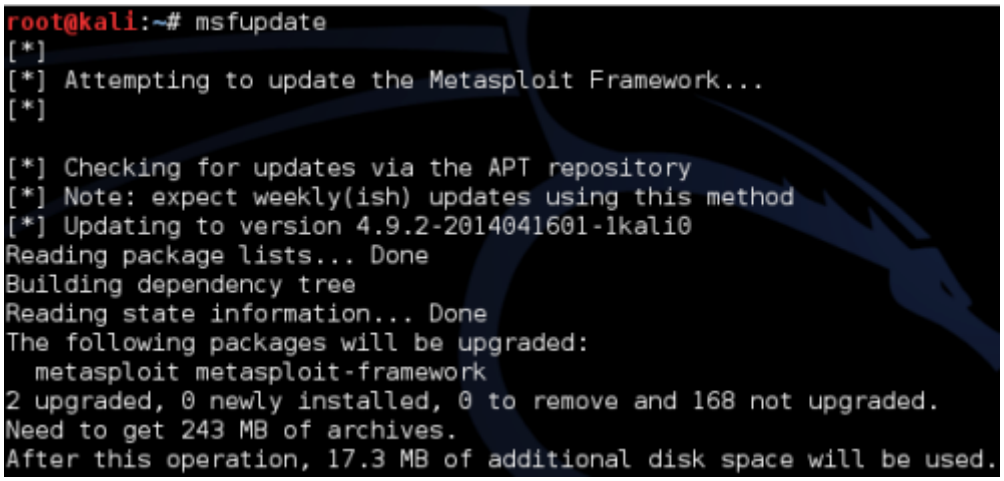

Figure 5. Updating Metasploit Framework

- Now run “msfconsole” to start Metasploit and you will be presented with the Metasploit console:

Figure 6. Metasploit Console

- Next search for the heartbleed modules by typing, “search heartbleed”:

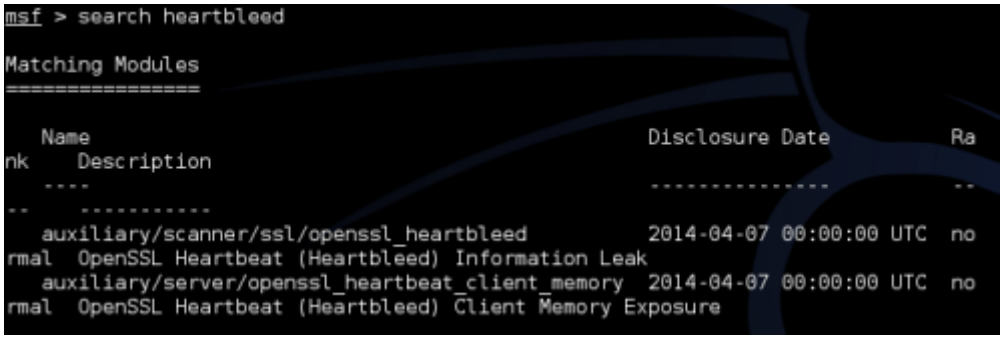

Figure 7. Using Metasploit Search Feature

Notice there are two, we will be using the scanner.

- Type, “use auxiliary/scanner/ssl/openssl_heartbleed”:

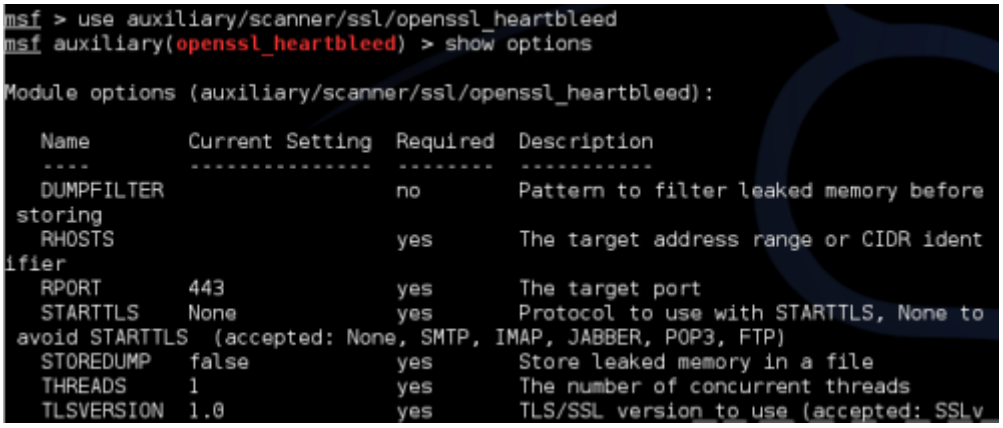

Figure 8. OpenSSL_Heartbleed Exploit Options

- We are just going to set two options, “set VERBOSE” to true and we need to “set RHOSTS” to our target IP address as seen in Figure 9:

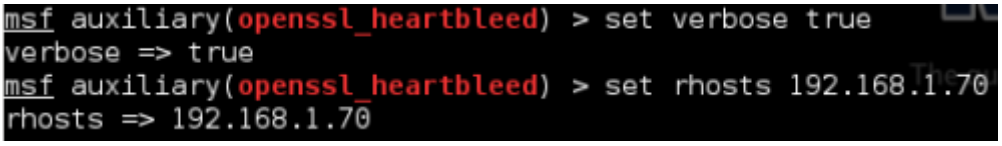

Figure 9. Setting Exploit Options

- And finally, just “run” the exploit:

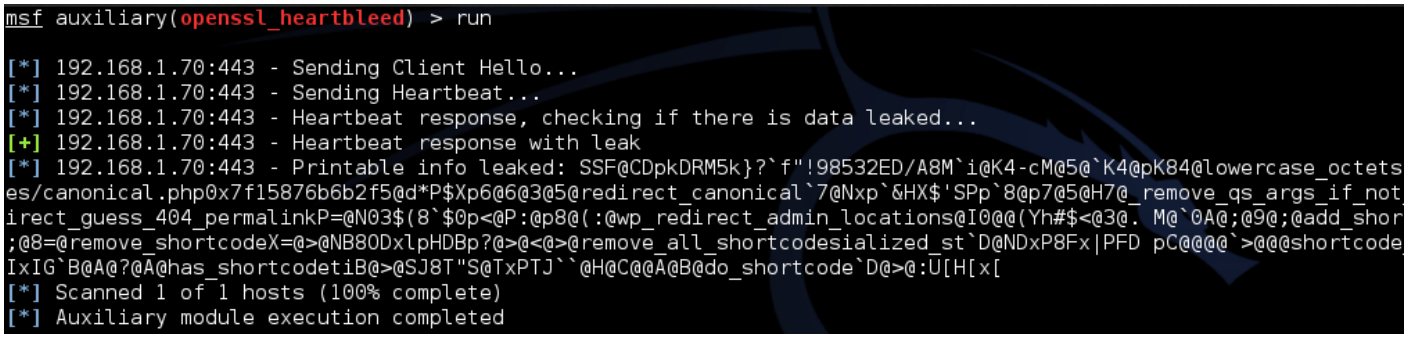

Figure 10. Random Data pulled from Server via Heartbleed

If you look at the picture above (Figure 10), you will see that Metasploit communicated with the server and was able to pull random data from the server’s memory.

If you re-run the module, you will get a totally different response from the server including different leaked data.

Conclusion

The important thing to note here is that the exploit pulls *random data* from memory. There is no guarantee that you will find account credentials, session cookie data or critical data every time you run this. The danger is in the fact that it could display sensitive data.

But be advised, with this type of attack it would be trivial to create a script that continuously hits a vulnerable site pulling data from it and then filter the responses for keywords like “username”, “password” or “cookie”.

Using a technique like this, the chance that confidential data could be recovered from server memory increases dramatically.

Thus the best practice (if you haven’t already) is to check your systems for the heartbleed vulnerability and patch them immediately. After all systems are patched, change any passwords on the effected machines.

References

1. *http://www.turnkeylinux.org/wordpress*
2. *http://www.offensive-security.com/kali-llnux-vmware-arm-image-download/*
3. *http://www.tenable.com/blog/tenable-facilitates-detection-of-openssl-vulnerability-using-nessus-and-nessus-perimeter*
4. *http://www.shodanhq.com*
5. *http://nmap.org/nsedoc/scripts/ssl-heartbleed.html*
6. *https://svn.nmap.org/nmap/scripts/ssl-heartbleed.nse*
7. *http://nmap.org/nsedoc/lib/tls.html*

About the Author

Daniel Dieterle is the founder of the “CyberArms Computer Security” blog (cyberarms.wordpress.com). He has 20 years of IT experience and is a technical editor for numerous books and training classes mainly focusing on Backtrack and Kali Linux. If you enjoyed this article and want to learn more about Kali Linux and Metasploit, check out Daniel’s latest book, “Basic Security Testing with Kali Linux” on Amazon.com.

Reversing Cracking

by Andreas Venieris

Everything published in this article is just for educational purposes and for "white" knowledge, that is the knowledge used only for defense. Respect the programmers' work. In general, use the knowledge you get from resources like this, to create more robust programs or better protecting tools.

According to Wikipedia: Software cracking is the modification of software to remove or disable features which are considered undesirable by the person cracking the software, usually related to protection methods: (copy protection, protection against the manipulation of software), trial/demo version, serial number, hardware key, date checks, CD check or software annoyances like nag screens and adware. (*http://en.wikipedia.org/wiki/Software_cracking*)

What You Will See

In this article you will learn what is required in order to start thinking as a cracker. I will show the least elementary steps needed for cracking simple programs.

I am going to give you two examples of how to crack a "home-made" program just to take the idea of what is behind in both parts: the cracker and the programmer. Next, I will show how to crack a small and simple commercial program. This program is no longer available (at least in the version that it was when it was cracked).

In general, you will learn:

- What we think when we crack.
- What knowledge we have to have.
- What tools we can use.

What you should know

- Basic knowledge of Assembly language (*http://en.wikipedia.org/wiki/Assembly_language*).
- An idea of how debuggers and disassemblers work and specially Olly debugger (Figure 1 and *http://www.ollydbg.de/*).
- Hex editors (you have to know at least what it is – *http://en.wikipedia.org/wiki/Hex_editor*).
- Basic knowledge of Machine Registers (*http://en.wikipedia.org/wiki/Processor_register*).
- Also, welcome (and many times required) is the knowledge of PE executables internal structure (*http://en.wikipedia.org/wiki/PE_executables*).
- Programming skills are always welcome ;)

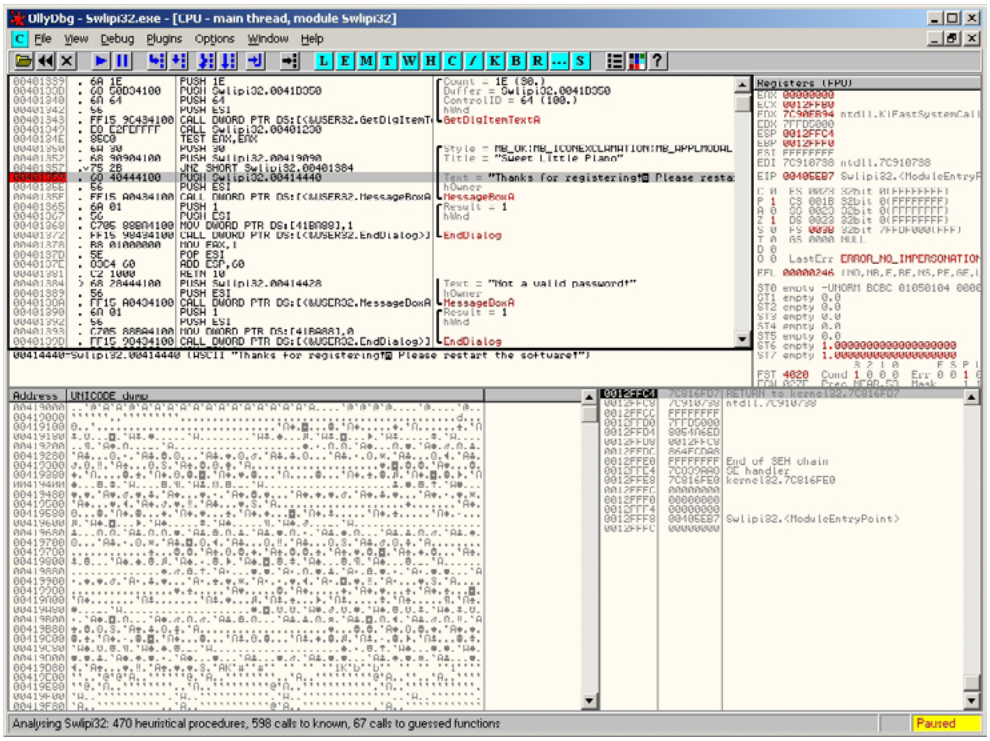

Figure 1. Olly: A well-known PE executable's debugger

Let's Crack

Everything published in this article is just for educational purposes and for "white" knowledge, that is the knowledge used only for defense. Respect the programmers' work. In general, use the knowledge you get from resources like this, to create more robust programs or better protecting tools.

Target #1: The "un-pressable" Button

Our first target is a program that programmers used to create for fun. A window appears and prompts the user to press a specific button, but when the mouse pointer comes over, the button becomes disabled (Figure 2).

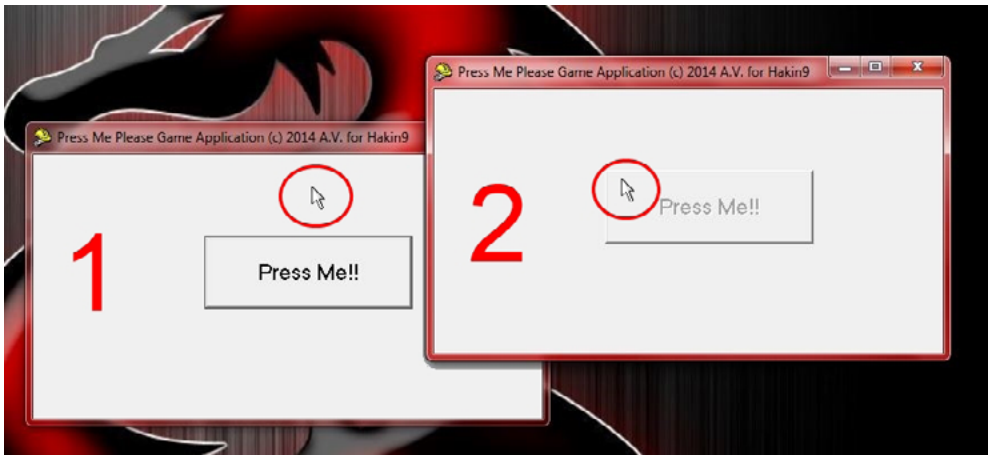

Figure 2. Press the button… if you can!

Sometimes, programmers used this trick when they realize that a user use the program in a not legitimate manner (unregistered, illegal copy etc.). Thus, the existed button "Start" (in every application's welcome screen) becomes disabled when the user try to press it!

The key here, for the cracker, is how to find a way to fool the application in order to allow the user to press the button before (or after…?) the program disable it. Here, we need some programming knowledge, a hex editor and (as always) some… tricky thoughts.

The method that we are going to use is the following: From elementary Windows programming we know that if we prefix any letter of any Caption Text Component with the ampersand symbol (&) the current letter becomes a shortcut to activate the component! Well…, this is the trick. We will change the caption “Press Me” of the main button of the application to “Press&Me”. This means that the user will see the caption as “PressMe” which means that the letter M is a shortcut for this button, which means that if he/she pressed the keys Alt+M is the same thing as he/she pressed the button itself.

The question now is how to change the caption of the button without access to the source code of the program. The answer is a Hex Editor. We open the executable program with a hex editor (Figure 3).

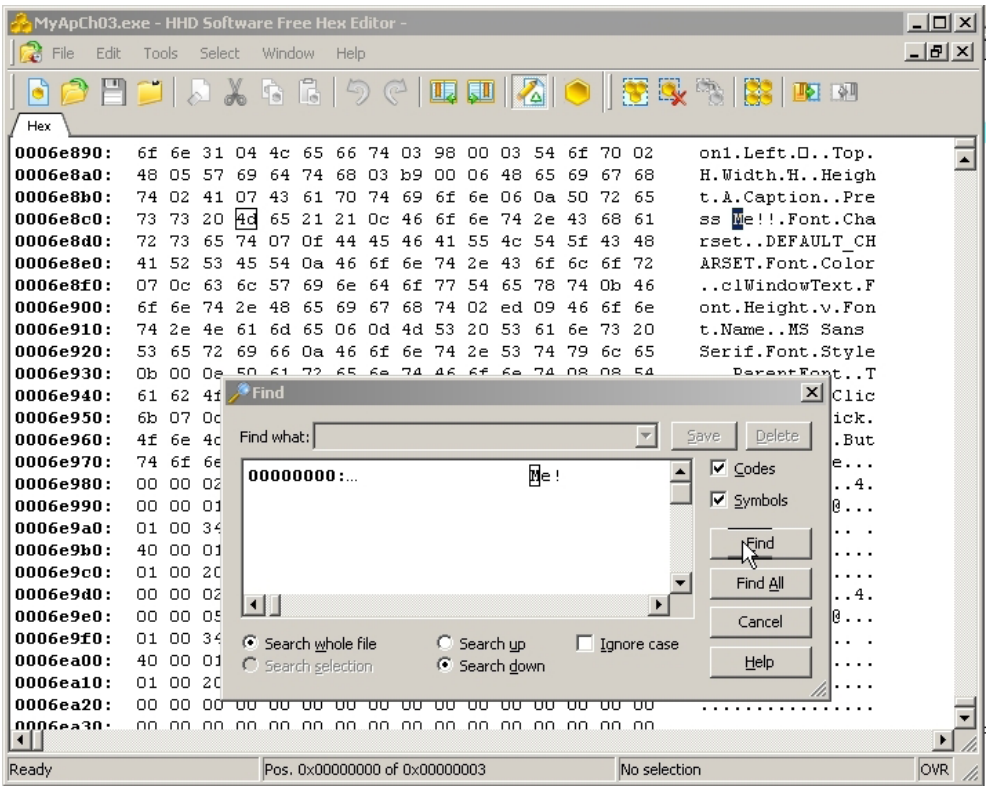

Figure 3. Using a hex-editor we can see the machine instruction of a program as hexadecimal numbers

What we see here can be divided into three parts that corresponds to the three columns that the cracker sees. The 1st column is the address of the current machine instruction, the 2nd column is the machine instruction itself in hexadecimal format and the 3rd column is (again) the machine instruction in raw text format or better, its text representation.

In Figure 3 you can see my first step, which is to find the text “Me!” in the text representation column. What is needed is to change the words “Press Me” (see the 3rd column) to “Press&Me”, and save the file (using the same name – or in a different name if you want to keep a backup).

Let's now run the program and press Alt+M (Figure 4). So easy, huh?

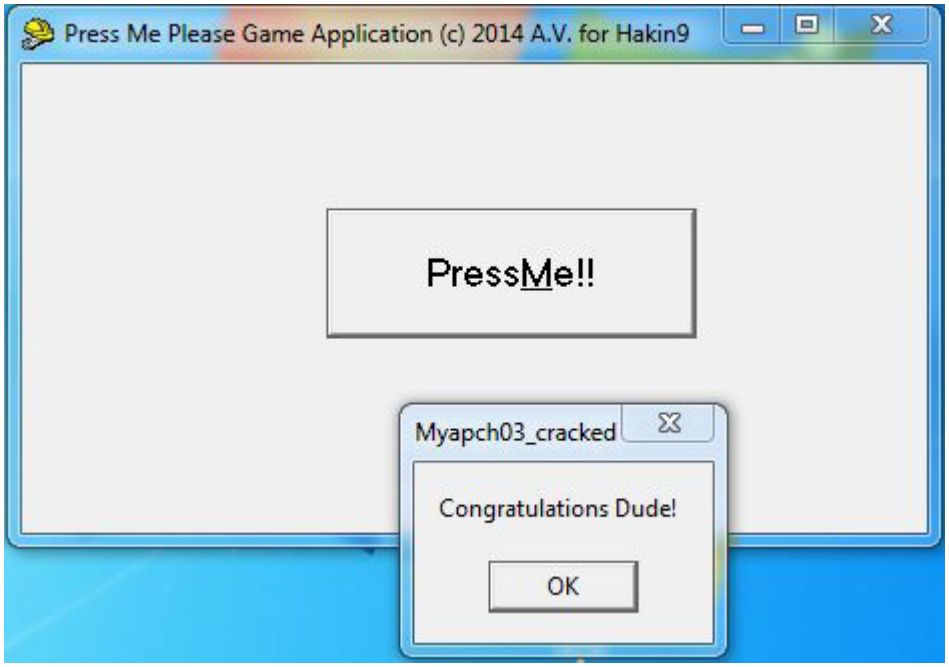

Figure 4. Press Alt+M and voila, the buttons is pressed without touching it ;)

Target #2: My Sweet Little Piano

It is time now to go a step further. We are going to bypass the registration control of a simple commercial application: the "Sweet Little Piano". The actual executable file is called *Swlipi32.exe*. This application converts our keyboard to a piano. The "problem" is that it requires a small amount of money in order to release all of its features.

When the program starts, asks for a password and (in addition) it displays a banner with some financial requirements of how to order, etc. (Figure 5).

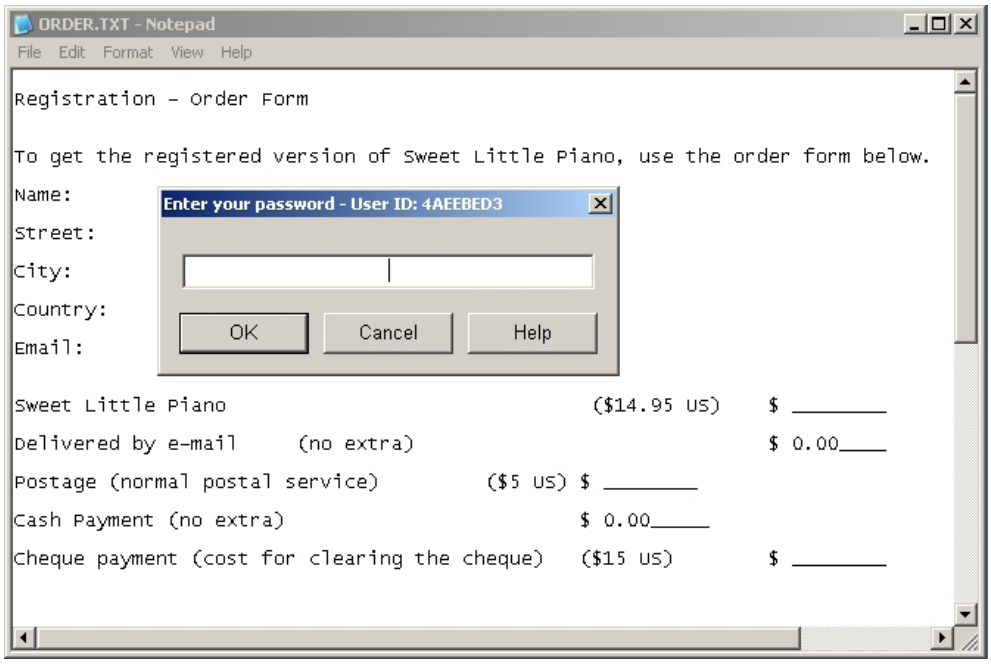

Figure 5. The welcome screen of the Sweet Little Piano

I press OK without entering any password, I get an error message: “Not a valid password!”. This message is the key of my method to bypass the protection. What I am going to do is:

- I will use the Olly Debugger to open the program.
- I find the machine instructions that correspond to the command that display the message “Not a valid password!”
- Near to the display command a check for a correct password must exist.
- I will modify the password check instruction in order to skip this check and…
- That’s it!

Ok, nice with the theory, let’s do the real thing now.

I use File | Open in Olly and I load the executable Swlipi32.exe. My first job then is to locate the string “Not a valid password!” (Figure 1). But wait, before proceed please let me explain to newcomers to Olly Debugger what we see in this Figure 1. The Olly window is divided into 4 parts:

- The top-left part shows the assembly instructions of the program.
- The top-right part displays the value of the machine registers.
- At bottom left and
- At Bottom right are the contents of the current running program’s memory (heap, stack) that we are not going to use (at least) in this article.

To find the string “Not a valid password!”, I right click on the Top Left Part of the Olly window and I select “Search For -> All Reference Text Strings”. Olly will respond with a new window with all machine instructions found containing the above string. The window has three columns: the Address, the Dissasembly and the Text String column. I am interested on the last one. So, I found the line (below):

```
Address             Disassembly              Text string
00401384            PUSH Swlipi32.00414428   ASCII "Not a valid password!"
```

I double Click on this line and Olly redirects me to the program’s corresponding instructions, here:

```
00401352   . 68 90904100    PUSH Swlipi32.00419090                  ; |Title = "Sweet Little Piano"
00401357     74 2B          JE SHORT Swlipi32.00401384
00401359   . 68 40444100    PUSH Swlipi32.00414440    ; |Text = "Thanks for registering! Please ...
0040135E   . 56             PUSH ESI                                ; |hOwner
0040135F   . FF15 A0434100  CALL DWORD PTR DS:[<&USER32.MessageBoxA>>; \MessageBoxA
00401365   . 6A 01          PUSH 1                                  ; /Result = 1
00401367   . 56             PUSH ESI                                ; |hWnd
00401368   . C705 88BA4100 >MOV DWORD PTR DS:[41BA88],1             ; |
00401372   . FF15 98434100  CALL DWORD PTR DS:[<&USER32.EndDialog>]  ; \EndDialog
00401378   . B8 01000000    MOV EAX,1
0040137D   . 5E             POP ESI
0040137E   . 83C4 60        ADD ESP,60
00401381   . C2 1000        RETN 10
00401384   > 68 28444100    PUSH Swlipi32.00414428        ; |Text = "Not a valid password!"
00401389   . 56             PUSH ESI                                ; |hOwner
0040138A   . FF15 A0434100  CALL DWORD PTR DS:[<&USER32.MessageBoxA>>; \MessageBoxA
00401390   . 6A 01          PUSH 1                                  ; /Result = 1
00401392   . 56             PUSH ESI                                ; |hWnd
00401393   . C705 88BA4100 >MOV DWORD PTR DS:[41BA88],0             ; |
0040139D   . FF15 98434100  CALL DWORD PTR DS:[<&USER32.EndDialog>]  ; \EndDialog
```

```
004013A3   . B8 01000000    MOV EAX,1
004013A8   . 5E             POP ESI
004013A9   . 83C4 60        ADD ESP,60
```

The line 00401384 (or better, the address) is responsible for displaying the message of password failure:

```
00401384   > 68 28444100    PUSH Swlipi32.00414428        ; |Text = "Not a valid password!"
```

The current message of failure could be the start of our success ;) If you check a few addresses above, at address 00401359, you can see a very interesting message: "Thanks for registering! Please... ".

```
00401359   . 68 40444100    PUSH Swlipi32.00414440    ; |Text = "Thanks for registering! Please ...
```

The program here *Thanks* the user for registering. It is very possible that very near a "jump to an address" is located that the program redirects the user, after a successful registration. Indeed there is such instruction, at address 00401357:

```
00401357      74 2B          JE SHORT Swlipi32.00401384
```

If case you wonder what the above instruction means in English then read this:

```
IF THIS = THAT THEN GOTO ADDRESS 00401384.
```

We know (from above) that the address 00401384 is the address of a wrong password. So, in pure English, we have the following situation here:

```
IF THE PASSWORD IS WRONG THEN DISPLAY ERROR MESSAGE.
```

I will make the following trick: I will change the above logic to:

```
IF THE PASSWORD IS CORRECT THEN DISPLAY ERROR MESSAGE.
```

This means that if the password is WRONG, the program will skip this instruction and will continue execution to the immediate next instruction, meeting the address 00401359 which is the address "Thanks for registering"! To do this I will change the instruction at address 00401357 from JE (Jump If Equal) to JNZ (Jump if Not Equal):

```
00401357      74 2B          JNZ SHORT Swlipi32.00401384
```

I am Double-Click to instruction "JE SHORT Swlipi32.00401384" and Olly displays a window with the current instruction. I just change the JE to JNZ and I press *Assemble*. This means that when the user enters a wrong password the programs will behave as it was correct! This situation has a funny side-effect: If we are so… "unlucky" and the password we enter is the real one, then we get an error message…! Well, in such case try another password and/or go to play in an online lottery as well… ;)

But, we are not finish yet! The registration form still appears when the application starts (even it gives us now all of its features). We should get rid of this annoying form. Hmm… if you are observant enough (and you should be if you want to be a good reverser) you will realize that the order form located in a file with the name *order.txt*. So, let's start search for the word "order.txt" in the Olly Debugger. I found it, at this address:

```
004018C0  /$ 8B4C24 04         MOV ECX,DWORD PTR SS:[ESP+4]
....
004018EC  |.^75 F7             \JNZ SHORT Copy_of_.004018E5
004018EE  |> 6A 01             PUSH 1                                   ; /IsShown = 1
004018F0  |. 8D5424 04         LEA EDX,DWORD PTR SS:[ESP+4]             ; |
004018F4  |. 52                PUSH EDX                                 ; |DefDir
004018F5  |. 6A 00             PUSH 0                                  ; |Parameters = NULL
004018F7  |. 68 30454100       PUSH Copy_of_.00414530                ; |FileName = "order.txt"
004018FC  |. 68 28454100       PUSH Copy_of_.00414528                   ; |Operation = "open"
```

```
00401901  |. 6A 00             PUSH 0                                   ; |hWnd = NULL
00401903  |. C64404 19 00      MOV BYTE PTR SS:[ESP+EAX+19],0           ; |
00401908  |. FF15 24424100     CALL DWORD PTR DS:[<&SHELL32.ShellExecuteA>]  ; \ShellExecuteA
0040190E  |. 81C4 68010000     ADD ESP,168
00401914  \. C3                RETN
```

The above code fragment is called *subroutine*. It is a set of one or more machine instructions that is called from one or more callers, i,e, memory addresses on the same program. The specific subroutine starts from address 004018C0. This means that somewhere in the program there is another machine instruction, the caller, in the form:

```
CALL Swlipi32.004018C0 (i.e. GOTO 004018C0).
```

Indeed if you search a bit we will find this:

```
0040277C     E8 3FF1FFFF    CALL Swlipi32.004018C0
```

What I want to do here is to make the program to ignore this instruction. But how? The answer to my problem is called (in assembly language terms) *NOP*, which means *No OPeration* (*http://en.wikipedia.org/wiki/NOP*). I am going to replace the current instruction with one or more NOPs. At the end, the instruction of the address `0040277C` will be:

```
0040277C   . 90               NOP
0040277D   . 90               NOP
0040277E   . 90               NOP
0040277F   . 90               NOP
00402780   . 90               NOP
```

In case that you wonder, why our initial one instruction becomes five instructions, please note that we must always keep the size of the initial instruction when we replace. Since each NOPs requires just 2 bytes and the initial instruction was 10 bytes long then we had to repeat the NOP, five times! Clever huh? To be honest, such specific tasks (nowadays) performed automatically by all good debuggers and of course Olly is one of them! Our final task is to save my changes to a new executable, the cracked one. So, I press *Right Click* | *Copy to executable* | *All Modifications*: A new window appear, I just press *Copy All*. A new window appears (again) that shows our modified assembly code. I close this window and a new one appears (again!! – I promise this is the last one!) that prompts me to enter the new executable filename. I enter "*Swlipi32_Cracked.exe*" and then OK! That's it! When I start the application I see the desired results (Figure 6).

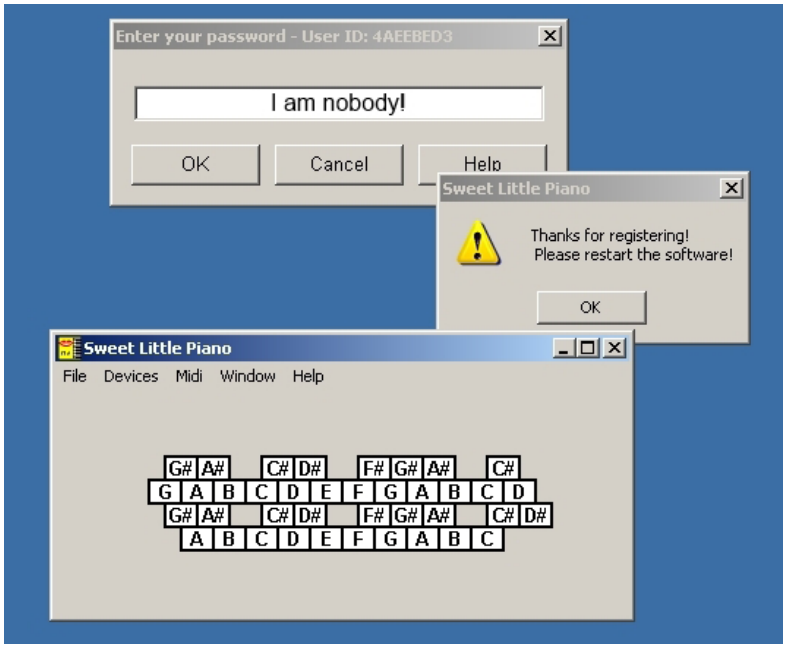

Figure 6. Our cracked Sweet Little Piano

Please note that this specific method in reverse engineering world is called *patching* (*http://en.wikipedia.org/wiki/Software_cracking*). It is considered as the most quick and dirty way to overcome a protection. From the "reversing science" point of view it is also considered as the less knowledgeable method, since it does not require a lot of effort (or knowledge). Nevertheless it is a required elementary step in order to be a good reverser.

As you see, we just scratch the iceberg's tip. Nowadays cracking is a lot more difficult even with the excellent cracking tools of the net or of the dark-net. Companies and programmers (hopefully!) are now more suspicious about cracking. New methods have been invented and new protections (packers, interrupt checkers, obfuscators etc). On the other hand, remember that what locks can be unlocked…

About the Author

Andreas holds a MSc in Artificial Intelligence from department of Computer Science from Queen Mary College – University of London and a BSc in Statistics from University of Piraeus. He was a PhD candidate with a scholarship in Department of Informatics of University of Piraeus in thesis "Architecture of an intelligent and secure database record".
He has contributed to the design and implementation of several management information systems for both commercial, industrial and web fields, for more than 10 years. He has worked in Software houses as well as Information Consultant Companies such as: ABC Professional Services (at Piraeus Bank), IMS Informatics (at Toyota Greece), Industrial Technologies SA, Greek Telecommunication Organization (OTE) and currently he is IT Director of Care Direct SA – a BTL Advertising & Digital Marketing Company. He is a member of the Economic Chamber Of Greece as well as the Greek Organization Of Scientists and Professionals of Informatics and Telecommunications. He likes programming and he is a computer security hobbyist and enthusiast.

Developing for Amazon Web Services?

Attend Cloud DevCon!

Cloud DevCon

June 23-25, 2014
San Francisco
Hyatt Regency Burlingame

www.CloudDevCon.net

Attend Cloud DevCon to get practical training in AWS technologies

- Develop and deploy applications to Amazon's cloud
- Master AWS services such as Management Console, Elastic Beanstalk, OpsWorks, CloudFormation and more!
- Learn how to integrate technologies and languages to leverage the cost savings of cloud computing with the systems you already have
- Take your AWS knowledge to the next level – choose from **more than 55 tutorials and classes,** and put together your own custom program!
- Improve your own skills and your marketability as an AWS expert
- Discover HOW to better leverage AWS to help your organization today

Register Early and SAVE!

A BZ Media Event

CloudDevCon

Dr.Web 9.0

for Windows — the rapid response anti-virus

1. Reliable protection against the threats of tomorrow
2. Reliable protection against data loss
3. Secure communication, data transfer and Internet search

www.drweb.com

Free 30-day trial: https://download.drweb.com

New features in Dr.Web 9.0 for Windows: http://products.drweb.com/9

FREE bonus — Dr.Web Mobile Security:
https://download.drweb.com/android